Part EIGHT
Concurrency

Chapter TWENTY-SIX
Thread Basics

Exam Objectives

Create worker threads using Runnable, Callable and use an ExecutorService to concurrently execute tasks.
Identify potential threading problems among deadlock, starvation, livelock, and race conditions.

Threads

In simple words, concurrency means doing things simultaneously, in parallel. In Java, concurrency is done with threads.

Threads are units of code that can be executed at the same time. They are sometimes called lightweight processes, although, in fact, a thread is executed within a process (and every process has, at least, one thread, the main thread).

At a low level, we can create a thread in two ways.

The first (and recommendable way) is to implement the `java.lang.Runnable` interface, which only has the `run()` method:

```
class RunnableTask implements Runnable {
    public void run() {
        System.out.println("Running");
    }
}
```

And then, to create a single thread, you have to pass an instance to the constructor of the `java.lang.Thread` class and **REQUEST** the thread to start (it might not start immediately. Besides, the order and time of execution of a thread are **NOT** guaranteed):

```
Thread thread = new Thread(new RunnableTask());
```

```
thread.start();
```

The `start()` method will call the `run()` method of the `Runnable` instance to start executing.

Of course, you can use an anonymous class to do it:

```
Thread thread = new Thread(new RunnableTask() {
    public void run() {
        System.out.println("Running");
    }
 });
thread.start();
```

Or since `Runnable` is a functional interface, a lambda expression:

```
Thread thread = new Thread(() -> {
    System.out.println("Running");
});
thread.start();
```

The other (discouraged) way is to subclass `Thread`, which implements `Runnable` so you just have to override `run()`:

```
class ThreadTask extends Thread {
    public void run() {
        System.out.println("Running");
    }
}
```

Then create an instance and call the `start()` method:

```
Thread thread = new ThreadTask();
thread.start();
```

However, it's better to implement `Runnable` on your own because with the new concurrency API, you don't have to create `Thread` objects directly anymore (not to mention that implementing an interface is the recommended object-oriented way to do it).

ExecutorService

Java 5 introduced a high-level API for concurrency, most of it implemented in the `java.util.concurrent` package.

One of the features of this API is the `Executor` interfaces that provide an alternative (better) way to launch and manage threads.

The `java.util.concurrent` package defines three executor interfaces:

Executor
This interface has the `execute()` method, which is designed to replace:

```
Runnable r = ... Thread t = new Thread(r);
t.start();
```

With:

```
Runnable r = ...
Executor e = ...
e.execute(r);
```

ExecutorService
This interface extends `Executor` to provide more features, like the `submit()` method that accepts `Runnable` and `Callable` objects and allows them to return a value.

ScheduledExecutorService
This interface extends `ExecutorService` to execute tasks at repeated intervals or with a particular delay.

Executors use thread pools, which use worker threads. These threads are different than the threads you create with the `Thread` class.

When worker threads are created, they just stand idle, waiting for work. When work arrives, the executor assigns it to the idle threads from the thread pool.

This way, threads are generic, they exist independently from the `Runnable` tasks they execute (in contrast to a traditional thread created with the `Thread` class).

One type of thread pool is the fixed thread pool, which has a fixed number of threads running. If a thread is terminated while it is still in use, it's automatically replaced with a new thread. There are also expandable thread pools.

Now, most of the time, you'd want to work with `ExecutorService`, since it has more functionality than `Executor`. Since they are interfaces, to create an instance of an `Executor` you have to use a helper class, `java.util.concurrent.Executors`.

The `Executors` class has many static methods to create an `ExecutorService`, like:

```
static ExecutorService newSingleThreadExecutor()
```

which creates an `Executor` that uses a single worker thread,

```
static ExecutorService
        newFixedThreadPool(int nThreads)
```

which creates a thread pool that reuses a fixed number of threads, and

```
static ScheduledExecutorService
        newScheduledThreadPool(int corePoolSize)
```

which creates a thread pool that can schedule task.

Let's start by creating an `ExecutorService` with a single thread pool:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
Runnable r = () -> {
    IntStream.rangeClosed(1, 4)
        .forEach(System.out::println);
};
System.out.println("before executing");
executor.execute(r);
System.out.println("after executing");
executor.shutdown();
```

A possible output can be:

```
before executing
after executing
1
2
3
4
```

Since there's only one thread (in addition to the main thread, don't forget that), tasks are guaranteed to be executed in the order they were submitted, and no more than one task will be active at any given time. In a real-world application, you may want to use `newFixedThreadPool()` with pool size equal to the number of processors available.

Once you're done with an `ExecutorService`, you have to shut it down to terminate threads and close resources. We have two methods to do it:

```
void shutdown()
List<Runnable> shutdownNow()
```

The `shutdown()` method tells the executor to stop accepting new tasks, but the previous tasks are allowed to continue until the finish. During this time, the method `isTerminated()` will return false until all tasks are completed, while the method `isShutdown()` will return true at all times.

The `shutdownNow()` method will also tell the executor to stop accepting new tasks but it will **TRY** to stop all executing tasks immediately (by interrupting the threads, but if the thread doesn't respond to interrupts, it may never terminate) and return a list of the tasks that were never started.

For example, if we execute:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
Runnable r = () -> {
    try {
        Thread.sleep(5_000);
    } catch (InterruptedException e) {
        System.out.println("Interrupted");
    }
};
executor.execute(r);
executor.shutdown();
```

We will have to wait around five seconds for the program to finish, but if we change `shutdown()` to `shutdownNow()`, `"Interrupted"` will be printed and the program will terminate immediately.

Besides `execute()`, there are other methods to submit a task. Let's define, first, the most critical methods and the new classes they use:

```
Future<?> submit(Runnable task)
```

Executes a `Runnable` and returns a `Future` representing that task.

```
<T> Future<T> submit(Callable<T> task)
```

Executes a `Callable` and returns a `Future` representing the result of the task.

```
<T> T invokeAny(
       Collection<? extends Callable<T>> tasks)
```

```
   throws InterruptedException, ExecutionException
```

Executes the given tasks, returning the result of one that has completed without throwing an exception, if any. The other tasks are canceled.

```
<T> List<Future<T>> invokeAll(
       Collection<? extends Callable<T>> tasks)
   throws InterruptedException
```

Executes the given tasks returning a list of `Future` objects holding their status and results when all complete (either normally or by an exception).

The `java.util.concurrent.Future` class has these methods:

```
boolean cancel(boolean mayInterruptIfRunning)
```

Attempts to cancel the execution of the task. If the argument is true, the thread is interrupted. Otherwise, the task is allowed to complete.

```
V get() throws InterruptedException, ExecutionException
```

Waits for the task to complete (indefinitely), and then retrieves its result.

```
V get(long timeout, TimeUnit unit)
    throws InterruptedException,
           ExecutionException,
           TimeoutException
```

Waits the given time at most and then retrieves the result (if the time is reached and the result is not ready, a `TimeoutException` is thrown).

```
boolean isCancelled()
```

Returns `true` if this task was canceled before it completed normally.

```
boolean isDone()
```

Returns `true` if the task is completed.

`java.util.concurrent.TimeUnit` is an `enum` with the following values:

- `TimeUnit.NANOSECONDS`
- `TimeUnit.MICROSECOND`
- `TimeUnit.MILLISECONDS`
- `TimeUnit.SECOND`
- `TimeUnit.MINUTES`
- `TimeUnit.HOURS`
- `TimeUnit.DAYS`

`java.util.concurrent.Callable` is a functional interface that defines this method:

```
V call() throws Exception;
```

As you can see, the difference with `Runnable` is that a `Callable` can return a value and throw a checked exception.

Now the code examples. When the `submit()` method is called with a `Runnable`, the returned `Future` object returns `null` (because `Runnable` doesn't return a result):

```
ExecutorService executor = Executors.newSingleThreadExecutor();
Runnable r = () -> {
    IntStream.range(1, 1_000_000).forEach(System.out::println);
};
Future<?> future = executor.submit(r);
try {
    // Blocks until the Runnable has finished
    future.get();
} catch (InterruptedException
    | ExecutionException e) { /** ... */ }
```

When this method is called with a `Callable`, the returned `Future` object contains the result when it has finished executing:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
Callable<Long> c = () ->
    LongStream.rangeClosed(1, 1_000_000).sum();
Future<Long> future = executor.submit(c);
try {
    // Blocks 1 second until the Callable has finished
    Long result = future.get(1, TimeUnit.SECONDS);
} catch (InterruptedException | ExecutionException
    | TimeoutException e) { /** ... */ }
```

Assuming the following list of `Callable` objects:

```
List<Callable<String>> callables = new ArrayList<>();
callables.add(() -> "Callable 1");
callables.add(() -> "Callable 2");
callables.add(() -> "Callable 3");
```

`invokeAny()` executes the given tasks returning the result of one that has completed successfully. You have no guarantee about which of the `Callable`'s results you'll get, just one of the ones that finish:

```
ExecutorService executor = Executors.newFixedThreadPool(3);
try {
    String result = executor.invokeAny(callables);
    System.out.println(result);
} catch(InterruptedException | ExecutionException e) {/** ... */}
```

Sometimes this will print `"Callable 1"`, sometimes `"Callable 2"`, and other times `"Callable 3"`.

`invokeAll()` executes the given tasks returning a list of `Future` objects that will hold the status and results until all tasks are completed. `Future.isDone()` returns `true` for each element of the returned list:

```
ExecutorService executor = Executors.newFixedThreadPool(3);
try {
    List<Future<String>> futures = executor.invokeAll(callables);
    for(Future<String> f : futures){
        System.out.format("%s - %s%n", f.get(), f.isDone());
    }
} catch(InterruptedException | ExecutionException e) { /** ... */ }
```

One possible output:

```
Callable 1 - true
Callable 2 - true
Callable 3 - true
```

Threading problems

When an application has two or more threads that make it behave in an unexpected way, well, that's a problem.

Generally, the cause of this issue is that threads are executed in parallel in such a way that sometimes they have to compete to access resources and other times, the actions of one thread will cause side-effects over the actions of another one (like modifying or deleting some shared values).

A solution to this problem is the concept of locking, where something like a resource or a block of code is locked with some mechanism in such a way that only one thread at a time can use or access it.

However, if we're not careful, locking can turn in one of the following three problems.

Deadlock
In simple words, a deadlock situation occurs when two or more threads are blocked forever, waiting for each other to acquire/release some resource.

For example, let's say you and I got into a big discussion, you say concurrency is hard to get right and I say it's easy. We're really mad at each other.

After a while, you're ready to say *"All right, it can be easy"* to end the discussion, but only if I say I'm wrong. In the same way, I'm willing to say that I'm wrong, that it is not easy, but only if you say it can be easy.

You hold the lock on *"It can be easy"* and you're waiting for me to release the lock on *"It's not easy."*

I'm holding the lock on *"It's not easy"* and I'm waiting for you to release the lock on *"It can be easy."*

But neither of us is going to admit that he/she is wrong (release the lock we have), because, who does that? So we are going to be waiting for each other (forever?). This is a deadlock situation.

Starvation
Starvation occurs when a thread is constantly waiting for a lock, never able to take it because other threads with higher priority are continually acquiring it.

Suppose you're in the supermarket, waiting in the checkout line. Then, a customer with a VIP membership arrives and is served first without waiting. And then, another VIP customer comes. And then another. And you just wait, forever. That's starvation.

Livelock
A livelock is like a deadlock in the sense that two (or more) threads are blocking each other, but in a livelock, each thread tries to resolve the problem on its own (live) instead of just waiting (dead). They are not blocked, but they are unable to make further progress.

Suppose we are walking in a narrow alley, in opposite directions. When we met, each of us moves aside to let the other pass, but we end up moving to the same side at the same time, repeatedly. That's a livelock.

There's another threading problem related with how concurrency works and some consider it the root cause of the previous problems.

Race condition
A race condition is a situation where two threads compete to access or modify the same resource at the same time in a way that causes unexpected results (generally, invalid data).

Let's say there's a movie you and I want to see. Each in our own house, we both go to the cinema's website to buy a ticket. When we check availability, there's only left. We both hurry and click the buy button at the same time. In theory, there can be three outcomes:

- Both of us get to buy the ticket
- Only one of us gets the ticket
- Neither of us gets the ticket (some other person wins it)

That's a race condition (in most race conditions there's a read and then a write).

The definite solution to this problem is never modify a variable (by making them immutable, for example). However, that's not always possible.

Another solution to avoid race conditions is to perform the read and write operations atomically (together in a single step). Another solution (also effective for the other problems) is to ensure the part where the problem happens is executed by only one thread at a time properly.

In the next chapter, we'll see how to implement these solutions with the concurrency API that Java provides.

Key Points

- At a low level, we can create a thread in two ways, either by implementing `Runnable` or by subclassing `Thread` and overriding the `run()` method.
- At a high-level, we use `Executor` s, which use thread pools, which in turn use worker threads.
- One type of thread pool is the fixed thread pool, which has a fixed number of threads running. We can also use single-thread pools.
- `ExecutorService` has methods to execute thread pools that either take a `Runnable` or `Callable` task. A `Callable` returns a result and throws a checked exception.
- The `submit()` method returns a Future object that represents the result of the task (if the task is a `Runnable` , `null` is returned).
- An executor has to be shutdown to close the pool thread with either `shutdown()` (gracefully) or `shutdownNow()` (forcefully).
- A deadlock situation occurs when two or more threads are blocked forever, waiting for each other to acquire/release some resource.
- Starvation happens when a thread is constantly waiting for a lock, never able to take it because other threads with higher priority are continually acquiring it.
- A livelock is like a deadlock in the sense that two (or more) threads are blocking each other, but in a livelock, each thread tries to resolve the problem on its own (live) instead of just waiting (dead).

- A race condition is a situation where two threads compete to access or modify the same resource at the same time in a way that causes unexpected results.

Self Test

1. Given:

```
ExecutorService service =
        Executors.newFixedThreadPool(2);
Future result = service.submit(() -> 1);
```

Assuming it compiles correctly, what is the type of the lambda expression?
A. `Runnable`
B. `Callable`
C. `Supplier`
D. `Function`

2. Which of the following statements are true?
A. When working with `Runnable`, you cannot use a `Future` object.
B. `Executor` implements `AutoCloseable`, so it can be used in a `try-with-resources`.
C. A `Callable` task can be canceled.
D. Thread pools contain generic threads.

3. Given:

```
Future future = executor.submit(callable);
future.get(3, TimeUnit.MILLISECONDS);
```

What does the `get()` method do?
A. It can return a value, after at most, 3 milliseconds. Otherwise, an exception is thrown.
B. It can return a value, after at most, 3 milliseconds. Otherwise, `null` is returned.
C. It returns a value after exactly 3 milliseconds.
D. It blocks the program for 3 milliseconds without returning anything.

4. Given:

```
public class Question_26_1 {
    private static Object A = new Object();
    private static Object B = new Object();
    public static void main(String[] args) {
       new Thread(() -> {
          acquireLock(A);
          System.out.println("Just acquired A");
          acquireLock(B);
          System.out.println("Just acquired B");
          releaseLock(B);
          releaseLock(A);
        }).start();
        new Thread(() -> {
          acquireLock(B);
          System.out.println("Just acquired B");
          acquireLock(A);
          System.out.println("Just acquired A");
          releaseLock(A);
          releaseLock(B);
        }).start();
    }
    private static void acquireLock(Objec o) {
        // Code to acquire lock on object o
    }
```

```
    private static void releaseLock(Objec o) {
        // Code to release lock on object o
    }
}
```

What threading problem is more likely to occur in this code?
A. Race Condition
B. Deadlock
C. Livelock
D. No problem at all

Open answers page

---

25. Files and Streams 27. Concurrency